# Enhancing Observability of Serverless Computing with the Serverless Application Analytics Framework

Robert Cordingly, Navid Heydari, Hanfei Yu, Varik Hoang, Zohreh Sadeghi, Wes Lloyd[†]
School of Engineering and Technology
University of Washington
Tacoma WA USA
rcording, navidh2, hanfeiyu, varikmp, zsadeghi, wlloyd@uw.edu

## ABSTRACT

To improve the observability of workload performance, resource utilization, and infrastructure underlying serverless Function-as-a-Service (FaaS) platforms, we have developed the Serverless Application Analytics Framework (SAAF). SAAF provides a reusable framework supporting multiple programming languages that developers can leverage to inspect performance, resource utilization, scalability, and infrastructure metrics of function deployments to commercial and open-source FaaS platforms. To automate reproducible FaaS performance experiments, we provide the FaaS Runner as a multithreaded FaaS client. FaaS Runner provides a programmable client that can orchestrate over one thousand concurrent FaaS function calls. The ReportGenerator is then used to aggregate experiment output into CSV files for consumption by popular data analytics tools. SAAF and its supporting tools combined can assess forty-eight distinct metrics to enhance observability of serverless software deployments. In this tutorial paper, we describe SAAF and its supporting tools and provide examples of observability insights that can be derived.

**ACM Reference Format:**
Robert Cordingly, Navid Heydari, Hanfei Yu, Varik Hoang, Zohreh Sadeghi, Wes Lloyd. 2021. Enhancing Observability of Serverless Computing with the Serverless Application Analytics Framework. In ACM/SPEC *International Conference on Performance Engineering Companion (ICPE '21 Companion), April 19-23, 2021, Virtual Event, France. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3447545.3451173*

## 1 Introduction

Recently serverless Function-as-a-service (FaaS) platforms have arisen that great simplify providing many of the desired features of distributed systems for software deployed to the cloud. FaaS platforms automate support for high availability, fault tolerance, and dynamic scaling, while billing developers only for the computational runtime of functions. Use of commercial serverless FaaS platforms, however, present challenges for profiling. These platforms frequently use virtual environments that restrict root access to the operating system, limit deployment package size, and lack a package manager making installation of existing profiling tools quite difficult. Observability of infrastructure is challenged given the abstract nature of FaaS platforms where hardware details and performance metrics are often hidden from the user. Finally, every commercial FaaS platform is different. Each platform supports different languages, supported by different backend infrastructure such as AWS Lambda's Firecracker MicroVM or Google Cloud Function's gVisor container application kernel [1,7].

To better understand performance implications of FaaS platforms, in this tutorial paper we introduce the Serverless Application Analytics Framework (SAAF) [2,8]. SAAF provides a cross-cloud, multi-language analytics framework. SAAF supports profiling functions written in Java, Python, Go, Node.js, and Bash on AWS Lambda, Google Cloud Functions, IBM Cloud Functions, Azure Functions, and OpenFaaS [9][10][11][12][5]. SAAF captures metadata regarding the performance, infrastructure, and resource utilization (e.g. CPU, memory, disk, and network) of software deployments made to Function-as-a-Service (FaaS) platforms. SAAF integration involves including the SAAF library in the function's package and invoking methods to specify profiling actions by adding a few lines of code. SAAF enables case studies and experiments on serverless application performance by supporting profiling and analytics of serverless software deployments. SAAF helps identify factors responsible for performance variation (e.g. alternate CPUs, function multitenancy, infrastructure freeze-thaw lifecycles) while enabling reproducible measurements of function performance across inherently heterogeneous infrastructure.

SAAF addresses an important problem with public cloud computing cyberinfrastructure in that it directly addresses the problem of observability on serverless cloud platforms. Cloud computing, and serverless platforms in particular, are known to abstract complexity and hide server implementation details from end users. Hiding information hinders efforts to reproduce performance measurements as sources of performance variation are obscured. Resource abstractions leveraged by serverless computing platforms also complicate the ability for practitioners to reconcile the costs of software deployments to the cloud. Cost reconciliation problems are increasing across public cloud platforms as cloud service delivery models becoming increasingly easy to use, but also more abstract. Factors such as resource heterogeneity and resource contention that are difficult to observe

*ICPE'21 Companion, April 19-23, 2021, Virtual Event, France.*
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8331-8/21/04…$15.00
https://doi.org/10.1145/3447545.3451173

can be responsible for producing considerable performance variation in the public cloud.

## 2 Serverless Application Analytics Framework

To enable profiling performance, resource utilization, and infrastructure of FaaS function executions, SAAF provides a library for inclusion inside the deployment package of each function [8]. Unlike frameworks that leverage proxy functions, or that are deployed directly on the host hardware of a FaaS platform, SAAF is integrated into the source code of the function to allow data collection from the function's perspective. This design enables SAAF to profile performance of function deployments on any commercial FaaS platform, while enabling introspection of the infrastructure used by each platform.

SAAF provides support tools to automate profiling experiments and data analysis. The FaaS Runner provides a multi-threaded client application that automates profiling experiments. FaaS Runner leverages a companion tool known as the ReportGenerator to then compile experiment results into reports that aggregate data for quick analysis. The ReportGenerator combines performance, resource utilization, and configuration metrics from many concurrent sessions enabling observations not possible when profiling individual FaaS functions calls.

| Platform | Python | Node.js | Java | Go | Bash |
|---|---|---|---|---|---|
| AWS Lambda | ✓ | ✓ | ✓ | ✓ | ✓ |
| Google Cloud Functions | ✓ | ✓ | ✗ | ✗ | ✗ |
| IBM Cloud Functions | ✓ | ✓ | ✓ | ✗ | ✗ |
| Azure Functions | ✓ | ✓ | ✗ | ✗ | ✗ |
| OpenFaaS | ✓ | ✗ | ✗ | ✗ | ✗ |

**Table 1: Currently Supported Platforms and Languages**

### 2.1 Supported Platforms Languages

SAAF provides support to profile functions created with Python, Node.js, Java, Go, and AWS Lambda custom runtimes using Bash. Each implementation is written natively in the respective language to offer the best performance, minimize dependencies, and to make using SAAF as easy as possible. Programmers include the SAAF library and a few lines of code to enable profiling. Documentation and example functions for each language are available from the SAAF GitHub repository [8]. Table 1 describes languages supported on each platform by SAAF.

SAAF includes publish scripts to help streamline the process of deploying functions to each platform. Projects can be built and deployed automatically to all supported platforms without requiring any code changes. This structure provides the ability to create multi-platform functions within a single code base.

### 2.2 Collecting Analytics with SAAF

SAAF collects metrics from the Linux procfs and appends them to the JSON payload returned by the function instance. Attributes collected include Linux CPU time accounting metrics such as CPU idle, user, kernel, and I/O wait time, wall-clock runtime, and memory usage [3]. As SAAF is dependent on Linux, SAAF does not support profiling resource utilization of functions deployed to Azure Functions using Windows. To identify infrastructure state, SAAF stamps function instances with a unique ID and uses the existence of the ID to identify if the environment is new (cold) or recycled (warm) [4].

After including the SAAF package and initializing the Inspector object, the attributes collected are defined by which functions the programmer calls. CPU, memory, function instance, Linux, and platform profiling functions offer granular and customizable profiling. Profiling functions and key metrics produced by SAAF, the FaaS Runner, and ReportGenerator are described online [8]. In total 48 distinct metrics can be obtained.

Profiling is enabled within FaaS functions through modifications in five sections:

1. **Initialization:** Initialize the SAAF Inspector object at the start of the FaaS function.
2. **Inspection:** Call initial SAAF inspect functions such as inspectAll(), inspectCPU(), etc. to collect base values for metrics.
3. **Workload:** Implement function, this is where the implementation of the function should be.
4. **Inspect Deltas:** After function code is complete, call SAAF inspect delta functions, e.g. inspectAllDeltas(), inspectCPUDeltas(), to calculate resource utilization.
5. **Finalize:** Generate SAAF output by calling the finish() function. Return this object or append to an existing return object. If the function is asynchronous, save this object to external data storage for future retrieval.

### 2.3 SAAF Profiling Verbosity

Key to SAAF's design is the ability to minimize profiling overhead as a component of the overall runtime of FaaS function calls. This overhead is reported by SAAF using the `frameworkRuntime` attribute measured in milliseconds. To address profiling overhead, we decompose profiling operations with nine inspection methods to offer different degrees of profiling verbosity. These methods enable the programmer to control how much data is collected and returned. If a programmer is interested in obtaining only a specific type of profiling data (e.g. CPU), then only a subset can be requested. These methods include: `inspectContainer()`, `inspectPlatform()`, and `inspectLinux()`. Initializing profiling for a particular class of metrics is performed by calling: `inspectCPU()` or `inspectMemory()`. These methods capture initial values of Linux resource utilization counters. Counters are typically reset to zero when new virtual infrastructure is created. FaaS platforms reuse existing virtual infrastructure to avoid function cold start latency for repeated function calls. Each FaaS platform has its own proprietary policy for infrastructure creation and retention [6]. Delta functions are called at the end of the function to capture the change in resource utilization: `inspectCPUDelta()` and `inspectMemoryDelta()`. Finally, methods are also provided to profile all metrics: `inspectAll()` and `inspectAllDeltas()`.

### 2.4 Running Experiments with FaaS Runner

FaaS Runner provides a multi-threaded client application used to define and orchestrate experiments on FaaS Platforms that works in conjunction with SAAF. FaaS Runner dedicates a separate thread to each client session and can orchestrate more than 1,000 concurrent FaaS function calls. The average runtime of the FaaS function, and the desired number of concurrent function calls for the experiment should dictate the computer used to host the FaaS Runner. A local computer with high network latency can be used for experiments with low concurrency requirements or when the FaaS function exhibits a high average runtime. Alternatively, a cloud-based virtual machine with very low network latency and many virtual CPU cores can be used to support experiments requiring high client throughput (e.g. AWS EC2 c5.24xlarge w/ 96 vCPUs).

FaaS Runner has the ability to orchestrate multiple types of workloads from basic single function executions, to complex multi-function pipelines. FaaS Runner can execute functions sequentially or in parallel, synchronously or asynchronously, across all of SAAF's supported platforms, and through HTTP requests. FaaS Runner provides multiple options to configure function execution and report generation for FaaS experiments. These options can be defined directly using command line arguments, or by using JSON configuration files. FaaS Runner supports two types of configuration files: function files and experiment files. Function files are used to define how to access individual FaaS function endpoints. Experiment files are used to define the operations and inputs of an experiment, as well as how experimental output should be aggregated to generate CSV report output. Raw unprocessed results of individual FaaS function invocations are captured using separate JSON files and persisted to disk in an output directory specified for the experiment. When all of the function calls in an experiment finish, FaaS Runner automatically invokes the ReportGenerator to compile and aggregate experimental results into a report. The ReportGenerator is described in section 2.5.

During an experiment, FaaS Runner can dynamically adjust a function's memory reservation size to automate profiling over a range of sizes. Additionally, FaaS Runner can distribute fixed or alternate data payloads to function calls. Payloads can be rotated across a set of function calls sequentially (e.g. round-robin) or shuffled randomly. Data payloads can be specified directly in the experiment JSON files, or by specifying an input directory where the user uploads a set of one or more input files. Payload inheritance allows defining a base set of attributes, simplifying specification as only the parameters that change need to be defined repeatedly in the experiment JSON file.

FaaS Runner experiment definitions are portable to any client computer enabling experiments to be reproduced enhancing potential to confirm experimental results. Experiments are launched by invoking the FaaS Runner application through the command line. Experiments execute autonomously to completion with no required user interaction enabling headless operation.

In addition to FaaS Runner's features for single function execution, FaaS Runner is capable of running and orchestrating complex pipelines of multiple functions. Sets of functions and experiment files can be input into FaaS Runner where each function will execute in order across each thread as a pipeline. Transitions can be defined to provide instructions for how FaaS Runner should transpose data from one function's response into the request of another. Combining the transitions attribute with programmable transition functions allows complex pipelines to be orchestrated where function execution order can be adapted based on the data returned in responses.

## 2.5 ReportGenerator

FaaS Runner's ReportGenerator aggregates FaaS function output files to produce user friendly reports in CSV format for consumption by popular data analytics tools including R, SciKit learn, and classical spreadsheets. The ReportGenerator calculates summary metrics such as the sum or average over column(s) to aggregate data from sets of FaaS function calls with many output files. In addition, the ReportGenerator calculates client-side metrics such as latency by observing the round-trip time of function calls from the client's perspective and subtracting runtime reported by SAAF. The average latency is then calculated by aggregating results over an entire batch of function calls. Function tenancy, which is the number of functions hosted by the same cloud-based virtual infrastructure, can be determined by comparing the number of function calls sharing the same vmID attribute where chronological time of execution overlaps.

The ReportGenerator supports regenerating reports over archived data. Report settings can be reconfigured in the JSON experiment file to explore alternate configuration settings. Additionally, reports can be generated over data from asynchronous function workloads retrieved from cloud object storage systems after FaaS workloads complete. Reports can also be generated by processing JSON obtained from alternate clients.

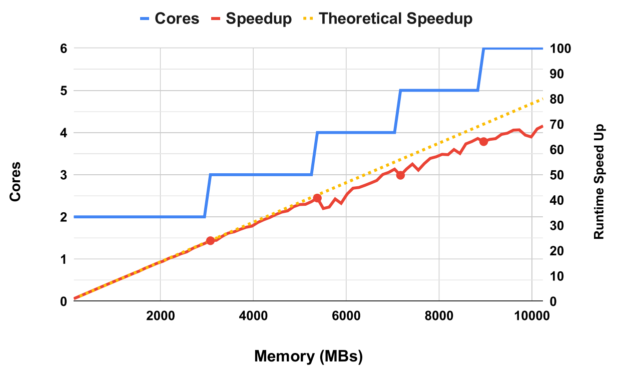

**Figure 1: AWS Lambda Performance Speedup for Sysbench Prime Number Generation vs. Function Memory**

## 3 Improving FaaS Observability With SAAF

Recently in November 2020, AWS Lambda expanded available memory for FaaS function execution to 10GB. To investigate implications for function performance we deployed Linux sysbench to calculate the first 2.5 million prime numbers. At 10GB, we anticipated that AWS Lambda would provide access to 6 vCPUs. To ensure we would saturate available CPU capacity, we configured sysbench to run using 12 threads. AWS Lambda states that for every doubling of memory, performance doubles.

We completed 80 sets of 100 runs for memory settings from 128MB to 10240MB incrementing by 128MB. Performance ranged from 275.6s @ 128MB to just 4.0s @ 10240MB, a speed-up of ~69.2x versus an expected theoretical speed-up of 80x. Figure 1 shows the observed vs. actual speedup and the available number of vCPUs identified by SAAF. Figure 2 depicts change is CPU utilization metrics for CPU user mode time, CPU kernel mode time, CPU idle time, CPU steal, and runtime as the function's memory is increased. This figure demonstrates how AWS Lambda adjusts the CPU timeshare relative to the function's memory size. CPU user mode time is seen as staying essentially constant as CPU idle time drops precipitously in a see-saw pattern as the function's available memory and CPU timeshare are increased.

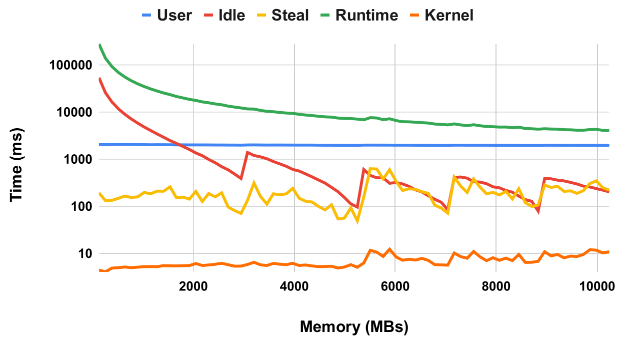

**Figure 2: Linux CPU Utilization (log scale) vs. Function Memory for Sysbench Prime Number Generation**

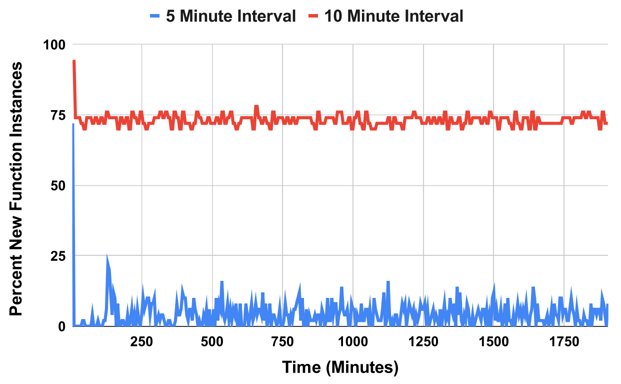

**Figure 3: AWS Lambda Function Instance Replacement vs. Function Call Interval over 24-hours**

As another example of observability, we depict the percentage of reused function instances for a Java-based prime number generator on AWS Lambda. We performed sets of 50 concurrent function calls to generate the first 100,000 prime numbers for 24 hours interspersing calls with a 5- or 10-minute delay. This workload creates 50 function instances, the runtime environments used to host individual function calls. Using SAAF's `newcontainer` attribute we visualize the percentage of new infrastructure created in Figure 3. A 10-minute call interval led to on average 73% replacement of infrastructure, versus just 4% replacement with a 5-minute interval where ~25% of calls replaced no infrastructure at all. Higher rates of infrastructure replacement will produce more function cold starts resulting in increased performance latency and longer response times.

## 4 Conclusions

SAAF provides a serverless computing profiling framework that enables insights into the performance and infrastructure of software deployments made to a variety of FaaS platforms in multiple languages. SAAF is easily integrated into new and existing functions deployed to commercial FaaS platforms. When combined with the FaaS Runner and ReportGenerator, SAAF provides an invaluable toolset for scientists and practitioners to automate and reproduce experiments to better evaluate performance tradeoffs of alternate serverless software designs. These insights enable developers to make educated design decisions to optimize function compositions, memory settings, or select the most performant programming language. Without these insights, developers often rely on ad hoc decisions without regard to the performance and cost implications.

## ACKNOWLEDGMENTS

Supported by the NSF Advanced Cyberinfrastructure Research Program (OAC-1849970), NIH grant R01GM126019, and AWS Cloud Credits for Research.

## REFERENCES

[1] Alexandru Agache, Marc Brooker, Andreea Florescu, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana Maria Popa. 2020. Firecracker: Lightweight virtualization for serverless applications. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020*.
[2] Robert Cordingly, Hanfei Yu, Varik Hoang, Zohreh Sadeghi, David Foster, David Perez, Rashad Hatchett, and Wes Lloyd. 2020. The Serverless Application Analytics Framework: Enabling Design Trade-off Evaluation for Serverless Software. In *WOSC 2020 - Proceedings of the 2020 6th International Workshop on Serverless Computing, Part of Middleware 2020*. DOI:https://doi.org/10.1145/3429880.3430103
[3] Wes J. Lloyd, Shrideep Pallickara, Olaf David, Mazdak Arabi, Tyler Wible, Jeffrey Ditty, and Ken Rojas. 2015. Demystifying the Clouds: Harnessing Resource Utilization Models for Cost Effective Infrastructure Alternatives. *IEEE Trans. Cloud Comput.* 5, 4 (2015), 667–680. DOI:https://doi.org/10.1109/tcc.2015.2430339
[4] Wes Lloyd, Shruti Ramesh, Swetha Chinthalapati, Lan Ly, and Shrideep Pallickara. 2018. Serverless computing: An investigation of factors influencing microservice performance. In *Proceedings - 2018 IEEE International Conference on Cloud Engineering, IC2E 2018*. DOI:https://doi.org/10.1109/IC2E.2018.00039
[5] Sunil Kumar Mohanty, Gopika Premsankar, and Mario Di Francesco. 2018. An evaluation of open source serverless computing frameworks. In *Proceedings of the International Conference on Cloud Computing Technology and Science, CloudCom*. DOI:https://doi.org/10.1109/CloudCom2018.2018.00033
[6] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *Proceedings of the 2020 USENIX Annual Technical Conference, ATC 2020*.
[7] Ethan G. Young, Pengfei Zhu, Tyler Caraza-Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2019. The true cost of containing: A gVisor case study. In *11th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2019, co-located with USENIX ATC 2019*.
[8] SAAF: Serverless Application Analytics Framework. Retrieved from https://github.com/wlloyduw/SAAF
[9] AWS Lambda - Serverless Compute. Retrieved from https://aws.amazon.com/lambda/
[10] Cloud Functions - Event-driven Serverless Computing. Retrieved from https://cloud.google.com/functions/
[11] IBM Cloud Functions. Retrieved from https://cloud.ibm.com/functions/
[12] Azure Functions - Develop Faster with Serverless Compute. Retrieved from https://azure.microsoft.com/en-us/services/functions/